Globus XIO Compression Driver: Enabling On-the-fly Compression in GridFTP

Mattias Lidman[1], John Bresnahan[2], Rajkumar Kettimuthu[1,3]
[1]Computation Institute, University of Chicago, Chicago, IL
[2]Redhat Inc.
[3]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
[*]kettimut@mcs.anl.gov

Abstract—**Multicore systems open the door to compression in Grid environments with high-speed networks to enable "faster than network speed" transfers. GridFTP is a data transport protocol that can break up its transfer payload in such a way that streaming it through multiple cores is possible. With the additional parallel processing power added by multicore systems it is possible to pipeline compression and packet switching in such a way that seemingly faster than network speed transfers are possible. In this paper, we present the Globus XIO compression driver, which enables GridFTP to compress data on-the-fly. We also present a detailed performance study of the compression driver using XIOPerf, an Iperf-like tool and GridFTP.**

Index Terms— **GridFTP, Compression, High-speed transfers**

I. Introduction

The need to move data faster is ever increasing. The use of compression to improve data transfer rates is not a new idea [1]. But this has not been adopted as a standard feature for data movement in Grid [2] environments. Also, for sites connected with by high-speed networks such as ESnet [3] and Internet2 [4], the network was not the bottleneck most of the time. But as host systems with many cores emerge, compression might provide benefits even on these high-speed networks. We have developed a compression driver for the Globus XIO framework [5] that compresses data as it is sent, and decompress, as it is received. The driver can under virtually any circumstances reduce network load by reducing the amount of data actually transferred. Under the right circumstances it can also help individual endpoints achieve higher-than-network speeds. This study will attempt to investigate under which circumstances this occurs, and when the driver is downright harmful. The answer to this question depends primarily on four factors:

• System resources - assuming infinite system resources and finite network bandwidth the driver will virtually always be beneficial or have no effect. In the case where the transferred data is completely random, the amount of data transferred will increase by a few hundreds of a percent. We will not consider this a factor because of the miniscule amount of possible overhead, and because of the relative rarity of transferring completely random data.

• Network bandwidth - assuming infinite bandwidth and finite system resources at the end-points, the driver will virtually always be a bottleneck to some extent.

• Block size - the driver compresses one block at a time and compression algorithms behave differently depending on block size.

• Data type - certain types of data (such as plain text) compresses nicely while others (such as random and already compressed data) do not.

We present a detailed performance study of the compression driver using XIOPerf [6], an Iperf-like tool and GridFTP [7,8], and provide a number of insights. The rest of the paper is organized as follows. In Section II, we provide background on GridFTP and Globus XIO. In Section III, we describe the compression driver. In Section IV, we present the experimental results and describe the conclusions drawn from the experiments in Section V. In Section VI, we discuss future work. We summarize in Section VII.

II. BACKGROUND

In this section we provide details on GridFTP and the Globus eXtensible Input/Output (XIO) framework.

A. GridFTP

The GridFTP protocol is a backward-compatible extension of the legacy RFC959 FTP protocol. It maintains the same command/response semantics introduced by RFC959. It also maintains the two-channel protocol semantics. One channel is for control messaging (the control channel), such as requesting what files to transfer and the other is for streaming the data payload (the data channel). Once a client successfully forms a control channel with a server, it can begin sending commands to the server. In order to transfer a file, the client must first establish a data channel. This task involves sending the server a series of commands on the control channel describing attributes of the desired data channel. Once these commands are successfully sent, a client can request a file transfer. At this point a separate data channel connection is formed using all the agreed-upon attributes, and the requested file is sent across it.

In standard FTP, the data channel can be used to transfer only a single file. Subsequent transfers must repeat the data channel setup process. GridFTP modifies this part of the protocol to allow many files to be transferred across a single data channel. This enhancement is known as data channel caching. GridFTP also introduces other enhancements to improve performance over the standard FTP mode. For example, parallelism and striping allow data to be sent over several independent data connections and reassembled at the destination.

Globus GridFTP is widely used to move large volumes of data over the wide area network. The XIO-based Globus GridFTP framework makes it easy to plug in other transport protocols. The Data Storage Interface [9] allows for easier integration with various storage systems. It supports non-TCP-based protocols such as UDT [10,11] and RDMA [12]. It also provides advanced capabilities such as concurrency [13] multilinking [14] and transfer resource management [15].

B. Globus XIO

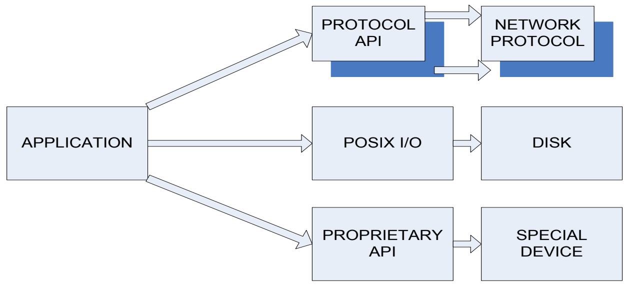

Figure 1. Typical application interaction with various devices.

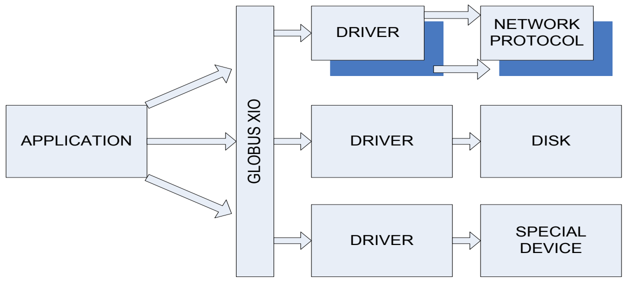

Figure 2. Application interaction with various devices via Globus XIO.

XIO is an extensible and flexible I/O library written for use with the Globus Toolkit. XIO is written in C programming language and provides us with one API that currently supports many different wire protocols. All implementations of these protocols are encapsulated as drivers that are modular.

GridFTP uses the XIO interface for network and disk I/O operations. The XIO framework presents a single, standard open/close, read/write interface to many different protocol implementations. The protocol implementations, called drivers, are responsible for manipulating and transporting the user's data. Drivers are grouped into a stack. When an I/O operation is requested, the XIO framework passes the operation request down the driver stack. An XIO driver can be thought of as a modular protocol interpreter that can be plugged into an I/O stack without concern about the application using it. This modular abstraction is what allowed us to achieve our success here without disturbing the application's tested code base and without forcing endpoints to run new and unfamiliar code.

III. Compression Driver

There are two types of drivers in XIO - transform drivers and transport drivers. Transport drivers are those that actually move data into or out of the process space. Examples of this are TCP and UDP. Transform drivers are those that manipulate, examine, frame, or change the data, or in other words, drivers that take any action other than moving the data across the process boundary. Examples of this are compression and logging.

The compression driver performs compression and decompression of data for XIO. It supports zlib and lzo compression algorithms. It is designed with ease of further development in mind - adding different methods of compressing data is straightforward. Like any XIO driver, the compression driver implements the open, close, read and write functions. The driver is designed in such a way that the code that handles various aspects of the communication with XIO and the code that manipulates the data are clearly abstracted. These are entirely separate entities; the block functionality does not know and does not care what the data handling functions does with the data, as long as they fulfill certain requirements. This separation makes it easier to add new compression strategies.

IV. Experimental Results

The experimental results were obtained using three different hardware configurations. Configuration 1: AMD Athlon 64 X2 3800+, Configuration 2: Intel Core2 Quad Q9300 Configuration 3: 2 Intel XEON 2.0GHz. In all configurations the hosts where connected by a 100Mbps local area network.

We used the following datasets for our experiments:

• ASCII - plain text. Original file consisted of 400MB of US census data. A common file type that is very compressible.

• Binary - consists of various software libraries. A common type of data that is somewhat compressible.

• MPEG - an MPEG encoded movie. This was used as an example of already compressed data, which will therefore be difficult to compress further.

• Random - a file consisting of random data created from /dev/urandom. This is used to establish a lower bound for compressible data.

• Zero - a file consisting entirely of zeros created from /dev/zero. This is used to establish an upper bound for compressible data.

Each file was 1.2GB in size. In the case of the ASCII and binary data, the original files were concatenated to fit this size.

The original files were several hundred MBs in size, which ensures that no unfair advantage was given to the compression algorithms due to the repetition of data.

The tests were performed using both the zlib and LZO compression libraries. In both cases the default compression level was used. The tests where done in a network environment which at times may have been used by others. To ensure that this did not affect the results, the tests where done during a time of year and times of day when the network could be expected to see minimal activity. Further, each test was run ten times and the results were taken as the average of each set of tests. In total, these results are the product of several days of network time.

The results were obtained using XIOPerf and GridFTP. The XIOPerf results were obtained using configuration 1 and 2. Configuration 3 was used for the GridFTP results. A brief description of XIOPerf is given below.

A. XIOPerf

XIOPerf, a network protocol testing and evaluation tool. XIOPerf is a command line program written on top of Globus XIO with a simple and well-defined interface to many different protocol implementations. XIOPerf was created to give users a way to quickly and easily experiment with an open-ended set of protocols over real networks to determine which will best suit their needs. XIOPerf presents a similar interface to that of IPerf. The main difference between IPerf and XIOPerf is that while IPerf is limited to TCP and UDP, XIOPerf is written on a framework that allows the user to plug in arbitrary protocol implementations.

B. Compression Ratios

Tables 1 and 2 show the compressed size of each type of data when transferred at a block size of 16KB for zlib and lzo respectively. In other words, this is the amount of data actually sent across the network when using the Compression Driver to transfer these 1.2GB files.

Table 1: Compression ratio for zlib

Type of data	Compressed size	Percentage of original file
Zero	1.72MB	0.14%
ASCII	97.04MB	8.15%
Binary	465.11MB	38.76%
MPEG	1172.49MB	97.70%
Random	1200.66MB	100.05%

Table 2: Compression ratio for lzo

Type of data	Compressed size	Percentage of original file
Zero	5.11MB	0.43%
ASCII	180.54MB	15.05%
Binary	599.31MB	49.94%
MPEG	1184.65MB	98.72%
Random	1204.99MB	100.42%

C. XIOPerf Results for zlib Compression

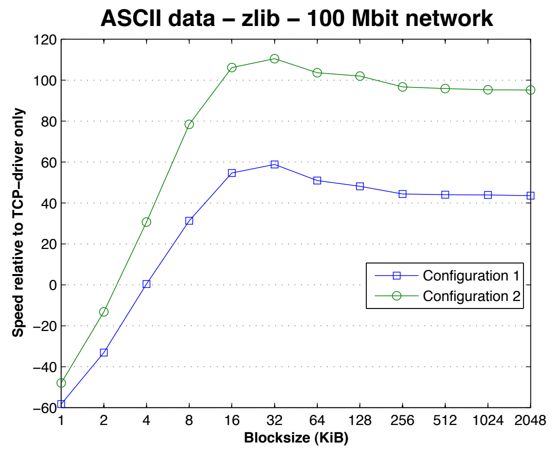

Figure 3: Performance comparison of compression driver (zlib) + TCP driver with TCP driver alone for ASCII data

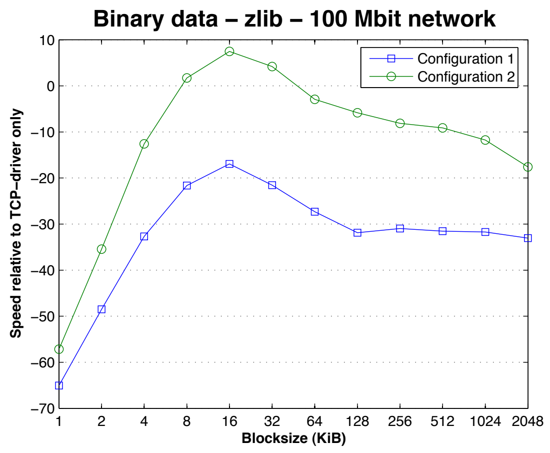

Figure 4: Performance comparison of compression (zlib) + TCP driver with TCP driver alone for Binary data

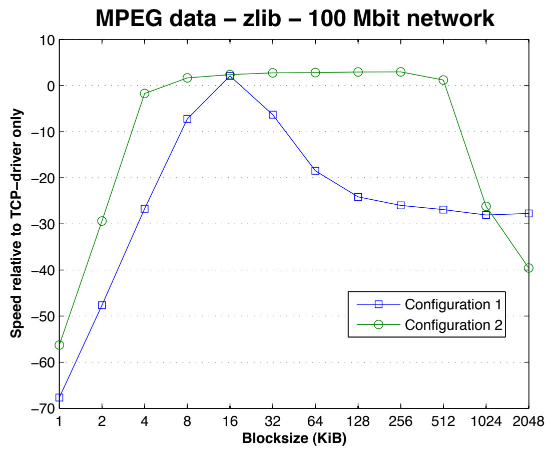

Figure 5: Performance comparison of compression (zlib) + TCP driver with TCP driver alone for MPEG data

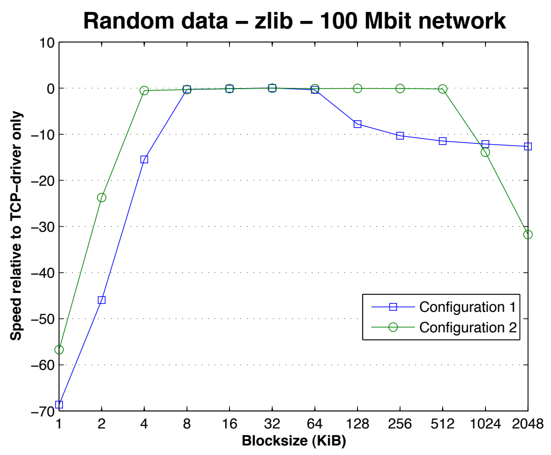

Figure 6: Performance comparison of compression (zlib) + TCP driver with TCP driver alone for Randon data

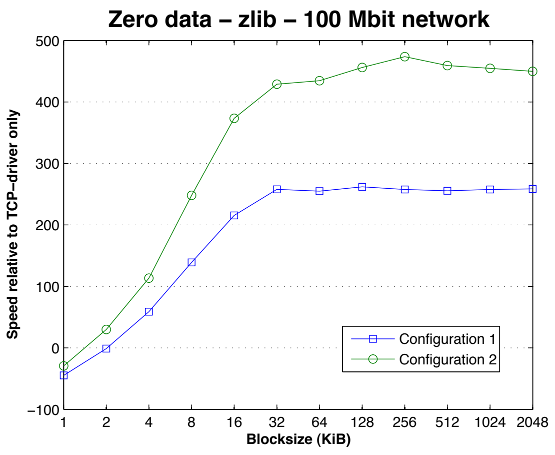

Figure 7: Performance comparison of compression driver (zlib) + TCP driver with TCP driver alone for Zero data

Figures 3 through 7 show the performance of using compression driver using zlib algorithm on top of the TCP driver in Globus XIO relative to that of using TCP driver alone, for various types of datasets. The results are mostly on the expected lines with zero files getting the big performance boost followed by ASCII. Interestingly, the performance for binary data is worse than random data especially for configuration 2.

D. XIOPerf Results for lzo Compression

Figures 8 through 12 show the performance of using compression driver using lzo algorithm on top of the TCP driver in Globus XIO relative to that of using TCP driver alone, for various types of datasets. The results are on the expected lines except that configuration 2 really did produce significantly lower results than configuration 1 at certain block sizes despite having much more powerful hardware.

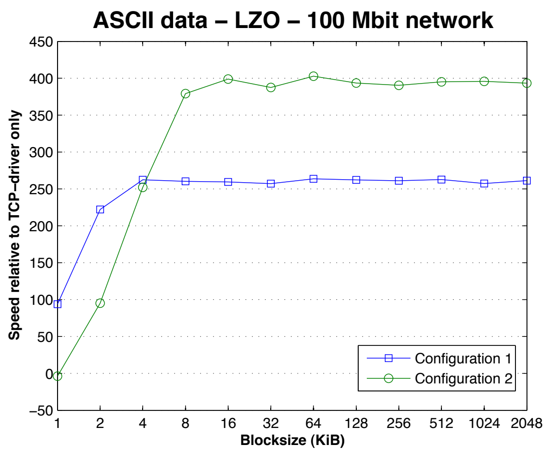

Figure 8: Performance comparison of compression driver (lzo) + TCP driver with TCP driver alone for ASCII data

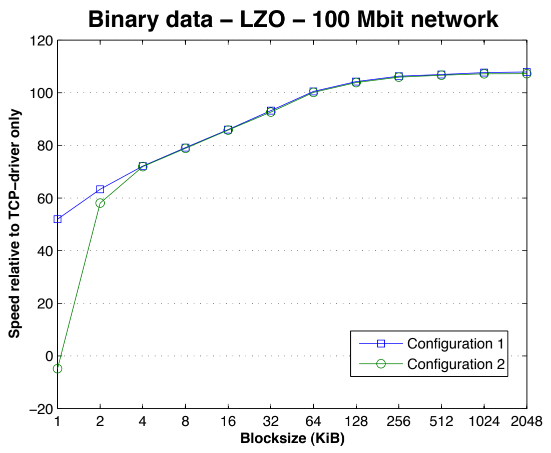

Figure 9: Performance comparison of compression driver (lzo) + TCP driver with TCP driver alone for Binary data

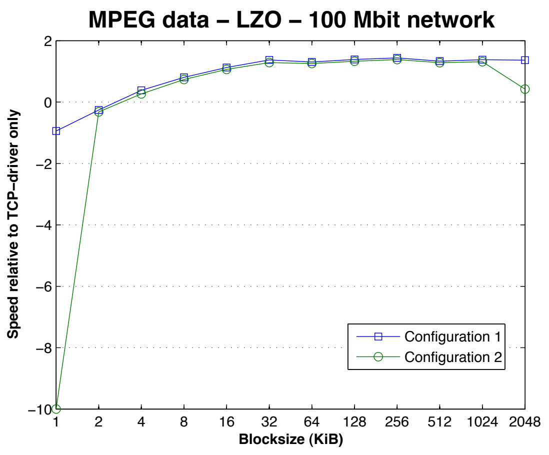

Figure 10: Performance comparison of compression (lzo) + TCP driver with TCP driver alone for MPEG data

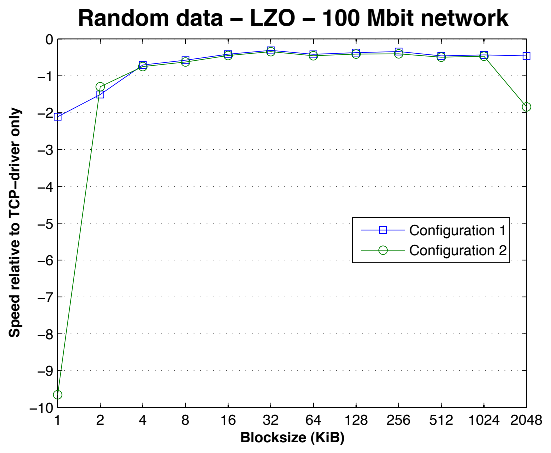

Figure 11: Performance comparison of compression (lzo) + TCP driver with TCP driver alone for Random data

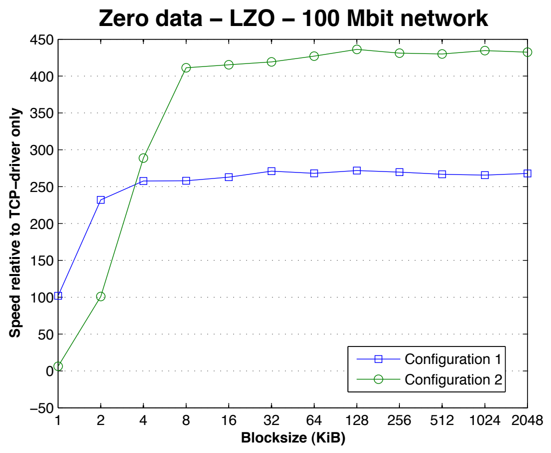

Figure 12: Performance comparison of compression (lzo) + TCP driver with TCP driver alone for Zero data

E. Compression Driver Overhead

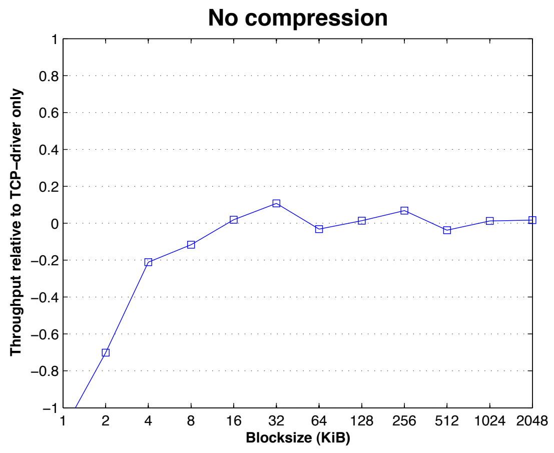

Figure 13: Compression driver overhead

The driver has a mode where data is transferred without being compressed or modified. This represents the minimum amount of overhead that will be added due to the presence of the driver, regardless of which compression method is used and to what extent the data is compressible. As can be observed from Figure 13, the driver itself introduces very less overhead – less than 0.2% for block sizes 4KB or more – exceeds 1% only at block size 1KB. Values above zero should be regarded as random fluctuations - the driver cannot increase performance other than by compressing data.

F. GridFTP results

One of the features of GridFTP is that it can make use of multiple parallel streams. As each stream is a separate thread, this driver should theoretically see an increase in performance when used in a multi-core/multi-CPU setup. To test this theory, these tests were performed with GridFTP using four different degrees of parallelism.

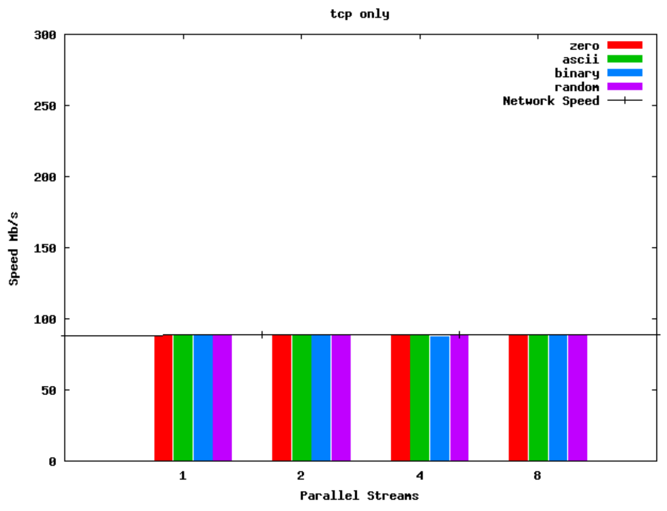

Figure 14: Performance of GridFTP with no compression

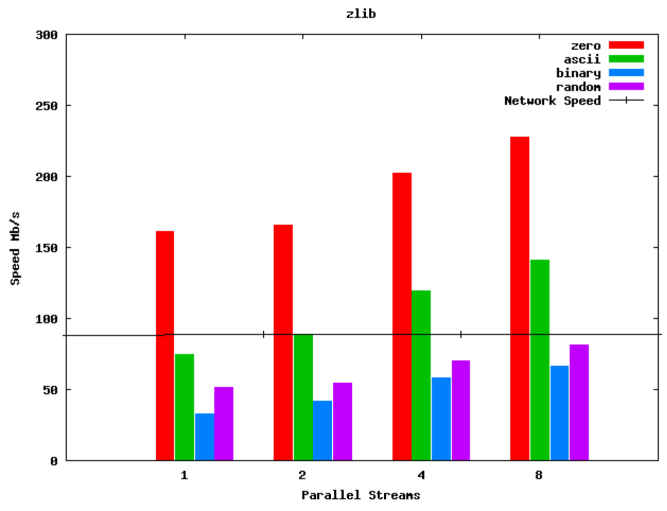

Figure 15: Performance of GridFTP with compression driver (zlib) for different parallelism values

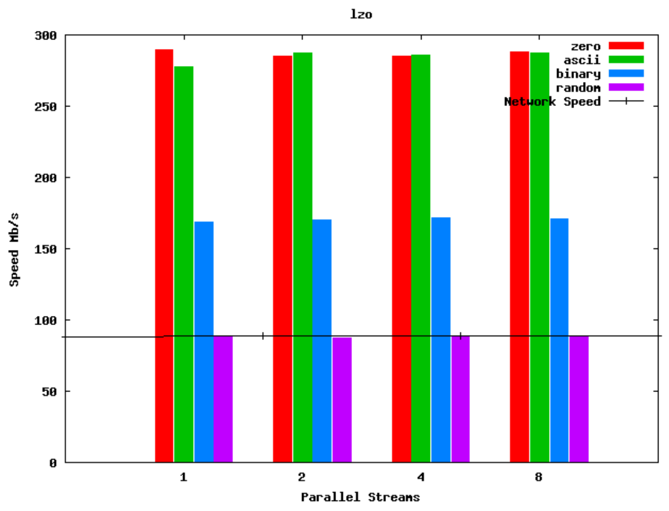

Figure 16: Performance of GridFTP with compression driver (lzo) for different parallelism values

G. Fluctuations in the Results

As noted above, each test was run ten times and the result was taken as the average. A general pattern could be noted that the fluctuations would be greater for highly compressible data. To get a feel for the these fluctuations, Table 3 presents the differences found in the results for ASCII and MPEG data using zlib-compression, at a block size of 16KB.

Table 3: Deviations in the results

	ASCII	MPEG
Mean value	185.10 m/s	91.92 m/s
Median value	187.16 m/s	91.94 m/s
Minimum value	176.03 m/s	91.79 m/s
Maximum value	192.30 m/s	91.95 m/s
Standard deviation	5.35 m/s	0.05 m/s

V. Interpreting the Results

A number of useful conclusions can be drawn by looking at the data that has been collected.

A. Effect on Network Load

The compression ratios measured tells us two important things about the performance of the driver. The first is the impact on network load. For example, when transferring the binary file used in these tests using zlib compression network load is reduced to 38.76%, to 97.70% for the MPEG, and so on. This means that in congested network environments using this driver may have a fortunate side effect – Increase the performance even for other applications and hosts by reducing total network load.

B. Effect on Host-to-Host Throughput

The second thing we learn looking at the compression ratios is the maximum change in speed the driver can achieve. We will refer to this as the optimal speed multiplier and is calculated by 1 / compression ratio.

Consider the following example, which matches the results for configuration 1 using zlib compression at a blocksize of 16KB:

• Throughput sans the compression driver is 90 Mbps.
• Throughput with the compression driver is 139 Mbps.
• Compression ratio is 0.0815.

If we could use hosts with infinite CPU resources, the observed speed would be roughly 90 * (1/0.0815) = 1104 Mbps. Of course, there is no such thing as a host with infinite CPU resources. But this is still useful knowledge, because it gives us an idea of when the local hosts will be the bottleneck, and when the network will. In this case the hosts are clearly bottlenecks by a wide margin since the observed result (about 139 Mbps) is a far cry from the theoretical maximum (1104 Mbps). Second, it let's us derive the average speed at which data is actually sent across the network by the formula 's/opt', where s is the observed speed when using the compression driver, opt is the optimal speed multiplier described above (inverse of the compression ratio). Continuing the above example, this means that the break-even point between the hosts and the network being the bottleneck for ASCII data occurs when the network allows a throughput of 139*0.0815 = 11.33 Mbps. Similarly, we can predict that the speed will remain fairly constant at around 139Mbps so long as network throughput remains above 11.33Mbps.

C. Effect of Data Type

The formula s/opt for describing at which network speed the local hosts ceases to be the bottleneck can also tell us conclusively that the more a particular type of data can be compressed, the more strain it puts on the local hosts: Above we concluded that the host in configuration 1 can only output 11.33 m/s of compressed ASCII data to the network. Consider the following example from the same test setup (configuration 1, 16KB block size, zlib compression), but transferring MPEG data, which is very difficult to compress further:

• Throughput sans the compression driver remains 90Mbps.
• Throughput with the compression driver is 91.70Mbps.
• Compression ratio is 0.977.

The formula s/opt yields that throughput should about 89.29Mbps - meaning that the CPU in this case can output compressed data essentially as fast as the network can transmit it. The results support this conclusion - configuration 2 is not able to exceed these results noticeably despite having a considerably more powerful processor.

D. Choice of the Compression Algorithm

In general, zlib offers the best compression while LZO beats zlib by a wide margin when it comes to speed. Which library is to be preferred depends on the situation - if the network is the bottleneck by a wide margin, zlib will likely be the better choice. If not, LZO will produce better results. LZO can be considered the safer choice, since it is less likely to have a negative impact. Also note that there is no great difference in speed between configuration 1 and 2 at most block sizes when using LZO, except for ASCII and zero data. Identical performance tells us that the host CPUs were not bottlenecks in either case, even without doing the type of calculations described above - if they were, configuration 2 would have showed higher performance than configuration 1 due to the more powerful CPU.

E. Effect of Block Size

As noted above, compression algorithms perform differently depending on how much data they are fed at a time - the block size. Generally, this means that a higher level of compression will be achieved with larger block sizes. But, as we have concluded, higher compression level does not automatically translate into higher speeds. So which block size is to be preferred? When using zlib-compression the optimal block size seems to be 16KB, or possibly 32KB. LZO seems to perform better when the block size is bigger, with the notable exception of the drop in performance for configuration 2 at the highest block size for random and MPEG data. No answer has been found for why this drop takes place. Also note that these results may vary depending on hardware configuration - we cannot conclude from these results that 16KB will be the optimal size for zlib for all configurations.

VI. Future Work

A. Customizing the Driver

Imagine you have a situation that matches that of configuration 1 using zlib compression - the driver is beneficial in some circumstances but detrimental in others. What you would want to do is add rules such as "only compress if block size is 4KB or larger, and if compression ratio is greater than 80%" which would mean that the driver would be detrimental under very few circumstances, if any. We plan to add the capability to allow this type of customization.

B. Dynamically Determine the Compression Strategy

We plan to add the ability to determine compression strategy dynamically. For example, do a test-compression of data using all available strategies and choose the one which gives the best ration. This should probably only be done once each transfer - if the first block compresses nicely, we can probably assume that the rest will also compress nicely.

VII. Summary

We have developed a compression driver for the Globus XIO framework and presented a detailed performance study using different compression techniques for different hardware configurations. We have also showed how this driver can be used to compress the data on-the-fly for GridFTP transfers and how it can be used to speed up the transfers using parallelism to take advantage of the multiple cores in the hosts. Our results indicate that consistently higher speeds can be achieved across a 100Mbps LAN without using state-of-the-art CPUs, as long as the data sent is somewhat compressible. LZO is the preferred compression library in most situations. More compressible data does not automatically result in a speed increase, since more compressible data is also more demanding on the CPU. While the actual compression puts significant strain on the CPU, the driver itself adds very little overhead.

References

[1] T. Chiueh, C. Yang, T. He, H. Pfister, A. Kaufman, "Integrated volume compression and visualization," Visualization '97 proceedings, vol., no., pp.329-336, 24-24 Oct. 1997

[2] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organization," The International Journal of High Performance Computing Applications, vol. 15, no. 3, pp. 200–222, Fall 2001.

[3] Energy Science Network – http://www.es.net

[4] Internet2 - http://www.internet2.edu/

[5] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link, "The Globus eXtensible Input/Output System (XIO): A Protocol Independent I/O System for the Grid," in Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 4, Vol. 5, IEEE Computer Society, Washington, DC, 2005. 179.1. DOI= http://dx.doi.org/10.1109/IPDPS.2005.429.

[6] J. Bresnahan, R. Kettimuthu, and I. Foster, "XIOPerf: A Tool for Evaluating Network Protocols," in Proceedings of the Third International Workshop on Networks for Grid Applications, 2006.

[7] Allcock, B., Bresnahan, J., Kettimuthu, R., Link, M., Dumitrescu, C., Raicu, I. and Foster, I., The Globus Striped GridFTP Framework and Server. SC'2005, 2005.

[8] R. Kettimuthu, L. Lacinski, M. Link, K. Pickett, S. Tuecke and I. Foster, Instant GridFTP. In 9th Workshop on High Performance Grid and Cloud Computing, May 2012.

[9] R. Kettimuthu, M. Link, J. Bresnahan, and W. Allcock, "Globus Data Storage Interface (DSI) – Enabling Easy Access to Grid Datasets," First DIALOGUE Workshop: Applications-Driven Issues in Data Grids, Aug. 2005.

[10] Y. Gu and R. L. Grossman, "UDT: UDP-based Data Transfer for High-Speed Wide Area Networks," Comput. Networks 51, no. 7 (May 2007), 1777–1799.

[11] J. Bresnahan, M. Link, R. Kettimuthu, I. Foster, "UDT as an Alternative Transport Protocol for GridFTP," 7th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT 2009), Tokyo, Japan, May 2009.

[12] H. Subramoni, P. Lai, R. Kettimuthu, D.K. Panda, "High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand," 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), May 2010.

[13] R. Kettimuthu, A. Sim, D. Gunter, B. Allcock, P. Bremer, J. Bresnahan, A. Cherry, L. Childers, E. Dart, I. Foster, K. Harms, J. Hick, J. Lee, M. Link, J. Long, K. Miller, V. Natarajan, V. Pascucci, K. Raffenetti, D. Ressman, D. Williams, L. Wilson, L. Winkler, "Lessons Learned from Moving Earth System Grid Data Sets over a 20 Gbps Wide-Area Network", 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010.

[14] J. Bresnahan, M. Link, R. Kettimuthu, I. Foster, "GridFTP Multilinking," 2009 TeraGrid Conference, Arlington, VA, June 2009.

[15] J. Bresnahan, M. Link, R. Kettimuthu, and I. Foster, "Managed GridFTP," 8th Workshop on High Performance Grid and Cloud Computing, May 2011